



UNIVERSITÀ DEGLI STUDI DI TRENTO

Dipartimento di Ingegneria e Scienza dell'Informazione

Corso di Laurea in
Informatica

ESAMI DI

PROGRAMMAZIONE FUNZIONALE

Prof.re Luca Abeni

Autore
Emanuele Nardi

Revisore
Matteo Contrini

Anno accademico 2017/2018

Introduzione

Lo scopo principale di questi appunti è quello di esaminare più da vicino gli esami di programmazione funzionale tenuti all'Università degli Studi di Trento. Queste note non sono complete, e la loro lettura non permette, da sola, di superare l'esame. La versione più recente di queste note si trova all'indirizzo:

<https://github.com/emanuelenardi/LaTeX-SML>

Il materiale didattico trattato in questo documento si trova nella cartella Google Drive del corso di Informatica [↗](#).

Hai trovato un errore? Inviami un'e-mail [↗](#) o contattami direttamente su Telegram [↗](#).

Ulteriori contatti dell'ateneo si trovano sulla pagina del DISI [↗](#).

Indice

Giugno 2015	5
Testo	5
Soluzione	5
Luglio 2015	7
Testo	7
Soluzione	7
Agosto 2015	8
Testo	8
Soluzione	8
Settembre 2015	9
Testo	9
Soluzione	9
Giugno 2016	10
1° Turno	10
Testo	10
Soluzione	10
2° Turno	10
Testo	10
Soluzione	10
Luglio 2016	12
1° Turno	12
Testo	12
Soluzione	12
2° Turno	13
Testo	13
Soluzione	13
Agosto 2016	14
Testo	14
Soluzione	14

Gennaio 2017	15
Testo mancante	15
Soluzione mancante	15
Febbraio 2017	16
Testo	16
Soluzione	16
Giugno 2017	17
1° Turno	17
Testo	17
Soluzione	17
2° Turno	17
Testo	17
Soluzione	17
Luglio 2017	19
1° Turno	19
Testo	19
Soluzione	19
2° Turno	19
Testo	19
Soluzione	20
Settembre 2017	21
Testo	21
Soluzione	21
Gennaio 2018	22
Testo mancante	22
Soluzione mancante	22

Lista dei Codici

1	Definizione di numero naturale tramite gli Assiomi di Peano	5
2	Definizione della funzione <code>somma</code> tramite gli Assiomi di Peano	5
3	Dichiarazione di numeri naturali	5
4	Definizione <i>alternativa</i> della funzione <code>somma</code> tramite gli Assiomi di Peano	5
5	Esempio di esecuzione	5
6	Definizione della funzione <code>prodotto</code> tramite gli Assiomi di Peano	6
7	Esempio di esecuzione	6
8	Definizione del tipo di dato <code>espressione Lambda</code>	7
9	Definizione della funzione <code>compute</code>	7
10	Definizione della funzione <code>elementi_pari</code>	8
11	Definizione del tipo di dato <code>codice</code>	9
13	Definizione della funzione <code>arriva</code>	9
14	Definizione della funzione <code>hist</code>	10
15	Definizione della funzione <code>noduplen</code>	10

16	Definizione del tipo di dato espressione Lambda	12
18	Definizione della funzione is_free	12
19	Definizione della funzione semplifica	13
20	Definizione del tipo di dato espressione	13
21	Definizione della funzione eval	13
22	Definizione del tipo di dato insiemidiinteri	14
23	Definizione della funzione vuoto	14
24	Definizione della funzione aggiungi	14
25	Definizione della funzione contiene	14
26	Definizione della funzione intersezione	14
27	Definizione della funzione unione	16
28	Definizione della funzione sommalì - 1° Turno	17
29	Definizione della funzione sommalì - 2° Turno	17
30	definizione della funzione eval	19
31	Esempio di ciclo for in C	19
32	definizione della funzione eval	20

Giugno 2015

Testo

Come noto, un numero naturale è esprimibile in base agli assiomi di Peano usando il seguente tipo di dato:

```
datatype naturale = zero | successivo of naturale;
```

Codice 1: Definizione di numero naturale tramite gli Assiomi di Peano

Usando tale tipo di dato, la somma fra numeri naturali è esprimibile come:

```
val rec somma = fn zero      => (fn n => n)
                | successivo a => (fn n => successivo (somma a n));
```

Codice 2: Definizione della funzione somma tramite gli Assiomi di Peano

Scrivere una funzione Standard ML, chiamata `prodotto`, che ha tipo `naturale -> naturale -> ↯ ↯ naturale`, che calcola il prodotto di due numeri naturali. Si noti che la funzione `prodotto` può usare la funzione `somma` nella sua implementazione.

Soluzione

Prendiamo confidenza con il tipo di dato definito:

```
> zero;
val it = zero: naturale

> successivo(successivo zero);
val it = successivo (successivo zero): naturale
```

Codice 3: Dichiarazione di numeri naturali

La somma fra numeri naturali è esprimibile in due modi, equivalenti fra loro, un modo è quello illustrato dal professore, l'altro è il seguente:

```
val rec somma = fn zero      => (fn n => n)
                | successivo a => (fn n => (somma a (successivo(n))));

val somma = fn: naturale -> naturale -> naturale
```

Codice 4: Definizione *alternativa* della funzione somma tramite gli Assiomi di Peano

Commento sulle due implementazioni:

Entrambe le definizioni di `somma` sono corrette. Nella prima definizione il caso *successivo a* restituisce una funzione che mappa una variabile `n` nel successivo della somma di `a` con `n`, nella seconda definizione, invece, il caso *successivo a* restituisce una funzione che mappa una variabile `n` nella somma di `a` con il successivo di `n`.

Il funzionamento dell'esecuzione della funzione `somma` fra due numeri naturali – definiti secondo gli Assiomi di Peano – è la seguente:

bisogna togliere un valore `successivo` al primo addendo affinché risulti pari al caso base (cioè zero). Questo lo si fa **o** aggiungendo un valore `successivo` alla somma del primo addendo con il secondo (1^a implementazione) **o** sommando il primo addendo con il successivo del secondo addendo (2^a implementazione).

```
> somma (successivo zero) (successivo (successivo zero));
val it = successivo (successivo (successivo zero)): naturale
```

Codice 5: Esempio di esecuzione

La somma di 1 e 2, risulta 3.

N.B. sono state aggiunte delle parentesi per far sì che gli argomenti dati in pasto alla funzione `somma` siano delle espressioni valutabili e non delle funzioni, quali sarebbero senza le parentesi.

```
val rec prodotto = fn zero      => (fn b => zero)
                  | successivo(a) => (fn b => (somma b (prodotto a b)));
```

```
val prodotto = fn: naturale -> naturale -> naturale
```

Codice 6: Definizione della funzione `prodotto` tramite gli Assiomi di Peano

Mostriamo un esempio di esecuzione della funzione `prodotto`:

```
datatype naturale = zero | succ of naturale;

val rec somma = fn zero      => (fn n => n)
                | succ a    => (fn n => succ (somma a n));

val rec prodotto = fn zero    => (fn b => zero)
                  | succ(a) => (fn b => (somma b (prodotto a b)));

prodotto (succ (succ (succ zero))) (succ (succ (succ zero)));
```

Codice 7: Esempio di esecuzione

Testo

Si consideri il seguente tipo di dato, che rappresenta una semplice espressione avente due argomenti x e y :

```
datatype Expr = X
                | Y
                | Avg of Expr * Expr
                | Mul of Expr * Expr
```

Codice 8: Definizione del tipo di dato espressione Lambda

dove il costruttore X rappresenta il valore del primo argomento x dell'espressione, il costruttore Y rappresenta il valore del secondo argomento y , il costruttore Avg , che si applica ad una coppia $(e1, e2)$, rappresenta la media (intera) dei valori di $e1$ ed $e2$, mentre il costruttore Mul (che ancora si applica ad una coppia $(e1, e2)$) rappresenta il prodotto dei valori di due espressioni $e1$ ed $e2$.

Implementare una funzione Standard ML, chiamata `compute`, che ha tipo `Expr -> int -> int -> int`.

Come suggerito dal nome, `compute` calcola il valore dell'espressione ricevuta come primo argomento, applicandola ai valori ricevuti come secondo e terzo argomento e ritorna un intero che indica il risultato finale della valutazione.

IMPORTANTE: notare il tipo della funzione! Come si può intuire da tale tipo, la funzione riceve tre argomenti usando la *tecnica del currying*. È importante che la funzione abbia il tipo corretto (indicato qui sopra). Una funzione avente tipo diverso da `Expr -> int -> int -> int` non sarà considerata corretta.

Soluzione

```
val rec compute = fn X      => (fn x => fn y => x)
                  | Y      => (fn x => fn y => y)
                  | Avg(e1, e2) => (fn x => fn y =>
                                     ((compute e1 x y) + (compute e2 x y)) div 2)
                  | Mul(e1, e2) => (fn x => fn y =>
                                     (compute e1 x y) * (compute e2 x y))

val compute = fn: Expr -> int -> int -> int
```

Codice 9: Definizione della funzione compute

Agosto 2015

Testo

Scrivere una funzione Standard ML, chiamata `elementi_pari`, che ha tipo `'a list -> 'a list`. La funzione riceve come parametro una α -lista e ritorna una α -lista contenente gli elementi della lista di ingresso che hanno posizione *pari* (il secondo elemento, il quarto elemento, etc...).

Per esempio

```
elementi_pari [1,5,2,10]
```

ritorna

```
[5,10]
```

Si noti inoltre che la funzione `elementi_pari` non deve cambiare l'ordine degli elementi della lista rispetto all'ordine della lista ricevuta come argomento (considerando l'esempio precedente, il valore ritornato deve essere `[5,10]`, non `[10,5]`).

Si noti che la funzione `elementi_pari` può usare i costruttori forniti da Standard ML per le alfa-liste, senza bisogno di definire alcun **datatype** o altro.

Soluzione

```
val rec elementi_pari = fn []      => []  
                        | [v]      => []  
                        | a::(b::l) => b::(elementi_pari l)
```

```
val elementi_pari = fn:'a list -> 'a list
```

Codice 10: Definizione della funzione `elementi_pari`

Testo

Si consideri il seguente tipo di dato:

```
datatype codice = rosso of string
                | giallo of string
                | verde of string;
```

Codice 11: Definizione del tipo di dato `codice`

che rappresenta un paziente in arrivo al pronto soccorso.

La stringa rappresenta il cognome del paziente, mentre i tre diversi costruttori `rosso`, `giallo` e `verde` rappresentano la gravità del paziente (codice `rosso`: massima gravità/urgenza, codice `verde`: minima gravità/urgenza).

Quando un paziente con codice `rosso` arriva al pronto soccorso, viene messo in lista d'attesa dopo tutti i pazienti con codice `rosso` (ma prima di quelli con codice `giallo` o `verde`); quando arriva un paziente con codice `giallo`, viene messo in lista d'attesa dopo tutti i pazienti con codice `rosso` o `giallo` (ma prima di quelli con codice `verde`), mentre quando arriva un paziente con codice `verde` viene messo in lista d'attesa dopo tutti gli altri pazienti.

Si scriva una funzione `arriva` (avente tipo `codice list -> codice -> codice list`) che riceve come argomenti la lista dei pazienti in attesa (lista di elementi di tipo `codice`) ed un paziente appena arrivato (elemento di tipo `codice`) e ritorna la lista aggiornata dei pazienti in attesa (dopo aver inserito il nuovo paziente nel giusto posto in coda).

Come esempio, l'invocazione

```
arriva [rosso "topolino", rosso "cip", giallo "ciop", verde "paperino", verde "pluto"] (↵
↵ giallo "clarabella");
```

deve avere risultato

```
[rosso "topolino", rosso "cip", giallo "ciop", giallo "clarabella", verde "paperino", ↵
↵ verde "pluto"]
```

IMPORTANTE: notare il tipo della funzione! Si noti inoltre che la funzione usa la *tecnica del currying* per gestire i due argomenti.

Soluzione

```
datatype codice = rosso of string
                | giallo of string
                | verde of string;
```

Codice 12: Definizione del tipo di dato `codice`

```
val rec arriva = fn
  []          => (fn x => [x])
| (verde n)::l => (fn (verde nn) => (verde n)::(arriva l (verde nn))
  | x          => x::((verde n)::l))
| (giallo n)::l => (fn (verde nn) => (giallo n)::(arriva l (verde nn))
  | (giallo nn) => (giallo n)::(arriva l (giallo nn))
  | x          => x::((giallo n)::l))
| (rosso n)::l => (fn x => (rosso n)::(arriva l x));
```

```
val arriva = fn: codice list -> codice -> codice list
```

Codice 13: Definizione della funzione `arriva`

Giugno 2016

1° Turno

Testo

Si scriva una funzione `hist` (avente tipo `real list -> real * real -> int`) che riceve come argomento una lista di `real` `l` ed una coppia di `real` `(c, d)`. La funzione `hist` ritorna il numero di elementi della lista compresi nell'intervallo `(c - d, c + d)`, estremi esclusi (vale a dire il numero di elementi `r` tali che $c - d < r < c + d$).

Come esempio, l'invocazione

```
hist [0.1, 0.5, 1.0, 3.0, 2.5] (1.0, 0.5);
```

deve avere risultato 1;

```
e hist [0.1, 0.5, 1.0, 3.0, 2.5] (1.0, 0.6);
```

deve avere risultato 2.

Soluzione

```
val rec hist = fn []      => (fn (c:real, d:real) => 0)
                | [e]     => (fn (c:real, d:real) =>
                                if (e > (c-d) andalso e < (c+d)) then
                                    1
                                else
                                    0)
                | (e :: l) => (fn (c:real, d:real) =>
                                if (e > (c-d) andalso e < (c+d)) then
                                    1 + hist l (c, d)
                                else
                                    0 + hist l (c, d));

val hist = fn: real list -> real * real -> int
```

Codice 14: Definizione della funzione `hist`

2° Turno

Testo

Si scriva una funzione `noduplen` (avente tipo `'a list -> int`) che riceve come argomento una lista di `'a` `l`. La funzione `noduplen` ritorna il numero di elementi della lista senza considerare i duplicati.

Come esempio, l'invocazione

```
noduplen ["pera", "pera", "pera", "pera"];
```

deve avere risultato 1;

```
noduplen ["red", "red", "green", "blue"];
```

deve avere risultato 3.

Soluzione

```
val rec noduplen = fn []      => 0
                    | [a]     => 1
                    | a::(b::l) => if (a <> b) then
                                    1 + noduplen (b::l)
                                else
                                    0
```

```
0 + noduplen (b::l);
```

```
val noduplen = fn: 'a list -> int
```

Codice 15: Definizione della funzione noduplen

Luglio 2016

1° Turno

Testo

Si consideri il tipo di dato

```
datatype lambda_expr = Var of string
                        | Lambda of string * lambda_expr
                        | Apply of lambda_expr * lambda_expr;
```

Codice 16: Definizione del tipo di dato espressione Lambda

che rappresenta un'espressione del Lambda-calcolo.

Il costruttore `Var` crea un'espressione costituita da un'unica funzione / variabile (il cui nome è un valore di tipo `string`); il costruttore `Lambda` crea una Lambda-espressione a partire da un'altra espressione, legandone una variabile (indicata da un valore di tipo `string`); il costruttore `Apply` crea un'espressione data dall'applicazione di un'espressione ad un'altra.

Si scriva una funzione `is_free` (avente tipo `string -> lambda_expr -> bool`) che riceve come argomenti una stringa (che rappresenta il nome di una variabile / funzione) ed una Lambda-espressione, ritornando `true` se la variabile indicata appare come libera nell'espressione, `false` altrimenti (quindi, la funzione ritorna `false` se la variabile è legata o se non appare nell'espressione).

Come esempio, l'invocazione

```
is_free "a" (Var "a")
```

deve avere risultato `true`, l'invocazione

```
is_free "b" (Var "a")
```

deve avere risultato `false`, l'invocazione

```
is_free "a" (Lambda ("a", Apply((Var "a"), Var "b")))
```

deve avere risultato `false`, l'invocazione

```
is_free "b" (Lambda ("a", Apply((Var "a"), Var "b")))
```

deve avere risultato `true` e così via.

IMPORTANTE: notare il tipo della funzione! La funzione usa la *tecnica del currying* per gestire i due argomenti.

Soluzione

```
datatype lambda_expr = Var of string
                        | Lambda of string * lambda_expr
                        | Apply of lambda_expr * lambda_expr;
```

Codice 17: Definizione del tipo di dato espressione Lambda

```
val rec is_free = fn s => fn Var v => s = v
                  | Lambda (v, e) => if (s = v) then
                                      false
                                      else
                                      is_free s e
                  | Apply (e1, e2) => (is_free s e1) orelse (is_free s e2);

val is_free = fn: string -> lambda_expr -> bool
```

Codice 18: Definizione della funzione `is_free`

2° Turno

Testo

Basandosi sul tipo di dato espressione e la funzione eval definiti come segue:

```
local
  val rec eval = fn costante    n      => n
                  | somma      (a1, a2) => (eval a1) + (eval a2)
                  | sottrazione (a1, a2) => (eval a1) - (eval a2)
                  | prodotto   (a1, a2) => (eval a1) * (eval a2)
                  | divisione  (a1, a2) => (eval a1) div (eval a2);
in
  val semplifica = fn costante    n      => costante(n)
                    | somma      (a1, a2) => costante((eval a1) + (eval a2))
                    | sottrazione (a1, a2) => costante((eval a1) - (eval a2))
                    | prodotto   (a1, a2) => costante((eval a1) * (eval a2))
                    | divisione  (a1, a2) => costante((eval a1) div (eval a2))
end;
```

(tipo funzione)

Codice 19: Definizione della funzione semplifica

il tipo espressione può essere esteso come segue per supporre il concetto di variabile:

```
datatype espressione = costante    of int
                        | variabile of string
                        | somma      of espressione * espressione
                        | sottrazione of espressione * espressione
                        | prodotto    of espressione * espressione
                        | divisione   of espressione * espressione
                        | var         of string      * espressione * espressione;
```

Codice 20: Definizione del tipo di dato espressione

Si riscriva la funzione eval per supportare i due nuovi costruttori `variabile` e `var`. `variabile x`, con `x` di tipo `string`, è valutata al valore della variabile di nome `x` (per fare questo, `eval` deve cercare nell'ambiente un legame fra tale nome ed un valore). `var (x, e1, e2)` è valutata al valore di `e2` dopo aver assegnato ad `x` il valore di `e1`.

Per poter valutare correttamente `variabile` e `var`, `eval` deve quindi ricevere come argomento l'ambiente in cui valutare le variabili. Tale ambiente può essere rappresentato come una lista di coppie (`string`, `intero`) ed avrà quindi tipo `(string * int)list`.

La funzione `eval` deve quindi avere tipo `(string * int)list -> espressione -> int`.

Soluzione

Questa è una possibile soluzione. Si noti che in questa soluzione la funzione `cerca` viene definita come visibile a tutti, mentre sarebbe più opportuno renderla locale a `eval` usando un costrutto `let` o `local`.

```
val rec cerca = fn s => fn [] => 0
                  | (s1, v)::l => if s1 = s then v else cerca s l;

val rec eval = fn env =>
  fn costante    n      => n
      | variabile  s      => cerca s env
      | somma      (a1, a2) => (eval env a1) + (eval env a2)
      | sottrazione (a1, a2) => (eval env a1) - (eval env a2)
      | prodotto   (a1, a2) => (eval env a1) * (eval env a2)
      | divisione  (a1, a2) => (eval env a1) div (eval env a2)
      | var        (v, e1, e2) => eval ((v, eval env e1)::env) e2;
```

Codice 21: Definizione della funzione eval

Testo

Si consideri una possibile implementazione degli insiemi di interi in standard ML, in cui un insieme di interi rappresentato da una funzione da `int` a `bool`:

```
type insiemediinteri = int -> bool;
```

Codice 22: Definizione del tipo di dato `insiemediinteri`

La funzione applicata ad un numero intero ritorna `true` se il numero appartiene all'insieme, `false` altrimenti. L'insieme vuoto è quindi rappresentato da una funzione che ritorna sempre `false`:

```
val vuoto:insiemediinteri = fn n => false;
```

```
val vuoto = fn: insiemediinteri
```

Codice 23: Definizione della funzione `vuoto`

ed un intero può essere aggiunto ad un insieme tramite la funzione `aggiungi`:

```
val aggiungi = fn f:insiemediinteri => fn x:int =>
  (fn n:int => if (n = x) then
    true
    else
    false
  ):insiemediinteri;
```

```
val aggiungi = fn: insiemediinteri -> int -> insiemediinteri
```

Codice 24: Definizione della funzione `aggiungi`

È possibile verificare se un intero è contenuto in un insieme tramite la funzione `contiene`:

```
val contiene = fn f:insiemediinteri => fn n:int => f n;
```

Codice 25: Definizione della funzione `contiene`

Si implementi la funzione `intersezione`, avente tipo `insiemediinteri -> insiemediinteri -> insiemediinteri`, che dati due insiemi di interi ne calcola l'intersezione.

IMPORTANTE: notare il tipo della funzione! Come si può intuire da tale tipo, usa la *tecnica del currying* per gestire i suoi due argomenti.

Soluzione

```
val intersezione = fn i1:insiemediinteri => fn i2:insiemediinteri =>
  (fn n =>
    ((contiene i1 n) andalso (contiene i2 n))
  ):insiemediinteri;
```

```
val intersezione = fn: insiemediinteri -> insiemediinteri -> insiemediinteri
```

Codice 26: Definizione della funzione `intersezione`

Gennaio 2017

Testo

Testo mancante.

Soluzione

Soluzione mancante.

Febbraio 2017

Testo

Si implementi la funzione `unione`, avente tipo `insiemediinteri -> insieme di interi -> insieme di interi`, che dati due insiemi di interi ne calcola l'unione.

Soluzione

```
val unione = fn i1:insiemediinteri => fn i2:insiemediinteri =>
  (fn n =>
    ((contiene i1 n) orElse (contiene i2 n))
  ):insiemediinteri;

val unione = fn: insieme di interi -> insieme di interi
```

Codice 27: Definizione della funzione `unione`

Giugno 2017

1° Turno

Testo

Si scriva una funzione `sommali` (avente tipo `int -> int list -> int`) che riceve come argomento un intero `n` ed una lista di interi `l`. La funzione `sommali` somma ad `n` gli elementi di `l` che hanno posizione *pari* (se la lista contiene meno di 2 elementi, `sommali` ritorna `n`).

Come esempio, l'invocazione

```
sommali 0 [1,2];
```

deve avere risultato 2;

```
sommali 1 [1,2,3];
```

deve avere risultato 3;

```
sommali 2 [1,2,3,4];
```

deve avere risultato 8.

Soluzione

```
val rec sommalì = fn z => fn []      => z
                    | v::[]      => z
                    | v1::v2::l => v2 + (sommali z l);
```

```
val sommalì = fn: int -> int list -> int
```

Codice 28: Definizione della funzione `sommali`

2° Turno

Testo

Si scriva una funzione `sommali` (avente tipo `int -> int list -> int`) che riceve come argomento un intero `n` ed una lista di interi `l`. La funzione `sommali` somma ad `n` gli elementi di `l` che hanno posizione *multipla di 3* (se la lista contiene meno di 3 elementi, `sommali` ritorna `n`).

Come esempio, l'invocazione

```
sommali 0 [1,2,3];
```

deve avere risultato 3,

```
sommali 1 [1,2,3];
```

deve avere risultato 4: e

```
sommali 2 [1,2,3,4,5,6];
```

deve avere risultato 11.

Soluzione

```
val rec sommalì = fn z => fn []      => z
                    | v::[]      => z
                    | v1::v2::[] => z
                    | v1::v2::v3::l => v3 + (sommali z l);
```

```
val sommali = fn: int -> int list -> int
```

Codice 29: Definizione della funzione sommali

Luglio 2017

1° Turno

Testo

Si consideri il tipo di dato `FOR = For of int * (int -> int)`; i cui valori `For(n, f)` rappresentano funzioni che implementano un ciclo for come il seguente:

```
int ciclofor (int x) {
    for (int i = 1; i < n; i++) {
        x = f(x);
    }
}
```

Si scriva una funzione `eval` (avente tipo `FOR -> (int -> int)`) che riceve come argomento un valore di tipo `FOR` e ritorna una funzione da interi ad interi che implementa il ciclo indicato qui sopra (applica $n - 1$ volte la funzione `f` all'argomento).

Come esempio, se `val f = fn x => x * 2`, allora `eval (For(3, f))` ritornerà una funzione che dato un numero `i` ritorna `i = 8`:

```
> val f = fn x => x * 2;
val f = fn: int -> int

> eval (For(3, f));
val it = fn: int -> int

> val g = eval (For(3, f));
val g = fn: int -> int

> g 5;
val it = 20: int
```

Soluzione

```
datatype FOR = For of int * (int -> int);

val rec eval = fn For (n, f) =>
    fn x => if (n > 0) then
        eval (For (n - 1, f)) (f x)
    else
        x;
```

Codice 30: definizione della funzione `eval`

2° Turno

Testo

Si consideri il tipo di dato `FOR = For of int * (int -> int)`; i cui valori `For(n, f)` rappresentano funzioni che implementano un ciclo for come il seguente:

```
int ciclofor (int x) {
    for (int i = 1; i < n; i++) {
        x = f(x);
    }
}
```

Codice 31: Esempio di ciclo for in C

Si scriva una funzione `eval` (avente tipo `FOR -> (int -> int)`) che riceve come argomento un valore di tipo `FOR` e ritorna una funzione da interi ad interi che implementa il ciclo indicato qui sopra (applica $n - 1$ volte la funzione `f` all'argomento).

Come esempio, se `val f = fn x => x * 2`, allora `eval (For(3, f))` ritornerà una funzione che dato un numero `i` ritorna `i = 4`:

```
> val f = fn x => x * 2;
val f = fn: int -> int

> eval (For(3, f));
val it = fn: int -> int

> val g = eval (For(3, f));
val g = fn: int -> int

> g 5;
val it = 20: int
```

Soluzione

```
datatype FOR = For of int * (int -> int);

val rec eval = fn For (n, f) =>
    fn x => if (n > 1) then
        eval (For (n - 1, f)) (f x)
    else
        x;
```

Codice 32: definizione della funzione `eval`

Settembre 2017

Testo

Si consideri il tipo di dato

```
datatype intonil = Nil | Int of int
```

ed una possibile implementazione semplificata di ambiente (che considera solo valori interi) basata su di esso:

```
type ambiente = string -> intonil
```

In questa implementazione, un ambiente è rappresentato da una funzione che mappa nomi (valori di tipo `string`) in valori di tipo `intonil` (che rappresentano un intero o nessun valore). Tale funzione applicata ad un nome ritorna il valore intero ad esso associato oppure `Nil`.

Usando questa convenzione, l'ambiente vuoto (in cui nessun nome è associato a valori) può essere definito come:

```
val ambientevuoto = fn _:string => Nil;
```

Basandosi su queste definizioni, si definisca una funzione `"lega"` con tipo `"ambiente -> string -> int -> ambiente"`. che a partire da un ambiente (primo argomento) genera un nuovo ambiente (valore di ritorno) uguale al primo argomento più un legame fra il nome e l'intero ricevuti come secondo e terzo argomento.

IMPORTANTE: notare il tipo della funzione! Come si può intuire da tale tipo, usa la *tecnica del currying* per gestire i suoi due argomenti.

A titolo di esempio:

- `((lega ambientevuoto "a"1)"a")` deve ritornare `Int 1`;
- `((lega ambientevuoto "a"1)"boh")` deve ritornare `Nil`;
- `((lega (lega ambientevuoto "a"1)"boh"~1)"boh")` deve ritornare `Int ~1`;
- `((lega (lega ambientevuoto "a"1)"boh"~1)"mah")` deve ritornare `Nil`.

Soluzione

```
val lega = fn e:ambiente =>
  fn nome =>
    fn valore =>
      (fn n => if (n = nome)
        then
          (Int valore)
        else
          (e n)): ambiente;
```

Gennaio 2018

Testo

Il testo dell'esame non è ancora stato rilasciato dal Prof.re. Nel caso l'avesse fatto ed io non avessi ancora aggiornato questo documento ti prego di contattarmi via e-mail [↗](#) o direttamente su Telegram [↗](#).

Soluzione